**C program to execute shell program and vice versa**

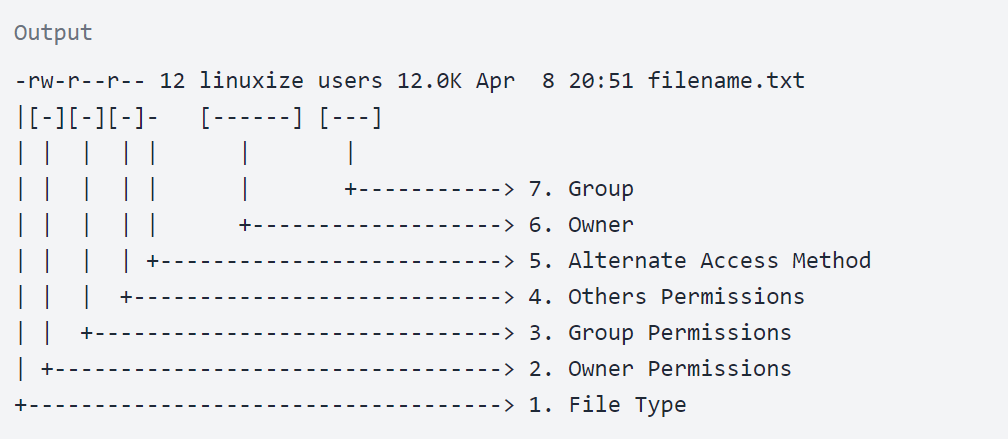
**Initial problem**

* Windows style newline characters can cause issues in Cygwin.
* The dos2unix command modifies newline characters so they are Unix / Cygwin compatible.
* It seems that you have Windows style line endings (\r\n) - you need to change them to unix style (\n). If you have dos2unix installed you could use it. You could also do it using sed or awk.
* Remove trailing \r character that causes this error:
* sed -i 's/\r$//' filename
* Option -i is for in-place editing, we delete the trailing \r directly in the input file. Thus be careful to type the pattern correctly.
* Assume we create a test.sh script. Note all the scripts would have the .sh extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct. For example −
* #!/bin/sh
* This tells the system that the commands that follow are to be executed by the Bourne shell. It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.

**Vim command**

https://opensource.com/article/19/3/getting-started-vim

**Chmod and permission and ls command**



* File types

1. “b” - Block special file.
2. “-“ - Regular file.
3. “c” - Character special file.
4. “d" - Directory.
5. “l” - Symbolic link.
6. “n” - Network file.
7. “p” - FIFO.
8. “s” - Socket.

* type of permissions

1. r - Permission to read the file.
2. w - Permission to write to the file.
3. x - Permission to execute the file.
4. s - setgid bit.
5. t - sticky bit.

* Type of users

1. u - The file owner.
2. g - The users who are members of the group.
3. o- All other users.
4. a - All users, identical to ugo

* The number 1 after the permission characters is the number of hard links to this file.
* The next two fields root root are showing the file owner and the group, followed by the size of the file (337), shown in bytes. Use the -h option if you want to print sizes in a human-readable format.
* The --sort option allows you to sort the output by extension, size, time and version:
* --sort=extension (or -X ) - sort alphabetically by extension.
* --sort=size (or -S) - sort by file size.
* --sort=time ( or -t) - sort by modification time.
* --sort=version (or -v) - Natural sort of version numbers.
* If you want to get the results in the reverse sort order, use the -r option.
* The -R option tells the ls command to display the contents of the subdirectories recursively:
* ls -l -s / ls – l- sh will show file size in human readable format
* chmod [OPTIONS] [ugoa…][-+=]perms…[,…] FILE...
* The first set of flags ([ugoa…]), users flags, by default is “a”.
* If no permissions are specified after the = symbol, all permissions from the specified user class are removed.
* Give the members of the group permission to read the file, but not to write and execute it:
  + chmod g=r filename
* Remove the execute permission for all users:
  + chmod a-x filename
* Repulsively remove the write permission for other users:
  + chmod -R o-w dirname
* Remove the read, write, and execute permission for all users except the file’s owner:
  + chmod og-rwx filename
* The same thing can be also accomplished by using the following form:
  + chmod og= filename
* Give read, write and execute permission to the file’s owner, read permissions to the file’s group and no permissions to all other users:
  + chmod u=rwx,g=r,o= filename
* Add the file’s owner permissions to the permissions that the members of the file’s group have:
  + chmod g+u filename
* chmod [OPTIONS] NUMBER FILE...
* When using the numeric mode, you can set the permissions for all three user classes (owner, group, and all others) at the same time.
* The NUMBER can be a 3 or 4-digits number.
* When 3 digits number is used, the first digit represents the permissions of the file’s owner, the second one the file’s group, and the last one all other users.
* Each write, read, and execute permissions have the following number value:
* r (read) = 4
* w (write) = 2
* x (execute) = 1
* no permissions = 0
* The permissions number of a specific user class is represented by the sum of the values of the permissions for that group.
* To find out the file’s permissions in numeric mode simply calculate the totals for all users classes. For example, to give read, write and execute permission to the file’s owner, read and execute permissions to the file’s group and only read permissions to all other users you would do the following:
* Owner: rwx=4+2+1=7
* Group: r-x=4+0+1=5
* Others: r-x=4+0+0=4

Using the method above we come up to the number 754, which represents the desired permissions.

* To recursively operate on all files and directories under the given directory, use the -R (--recursive) option:

chmod -R MODE DIRECTORY

* <https://linuxize.com/post/how-to-list-files-in-linux-using-the-ls-command/>
* <https://www.pluralsight.com/blog/it-ops/linux-file-permissions>
* <https://linuxize.com/post/chmod-command-in-linux/>

**Variable**

* The reason you cannot use other characters as variable name such as !, \*, or - is that these characters have a special meaning for the shell.
* Read-only Variables
* Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.
* For example, the following script generates an error while trying to change the value of NAME −

**#!/bin/sh**

**NAME="Zara Ali"**

**readonly NAME**

**NAME="Qadiri"**

* The above script will generate the following result − /bin/sh: NAME: This variable is read only.
* Unsetting Variables
* Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.
* Following is the syntax to unset a defined variable using the unset command –

**unset variable\_name**

* **readonly variable cannot be unset**
* Variable Types

When a shell is running, three main types of variables are present −

1. Local Variables − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
2. Environment Variables − An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
3. Shell Variables − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

**$ and command line argument**

* The command-line arguments $1, $2, $3, ...$9 are **positional parameters**, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

|  |  |
| --- | --- |
| **Sr.No.** | **Variable & Description** |
| 1 | **$0**   * The filename of the current script. |
| 2 | **$n**   * These variables correspond to the arguments with which a script was invoked. * Here **n** is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| 3 | **$#**   * The number of arguments supplied to a script. |
| 4 | **$\***   * All the arguments are double quoted. * If a script receives two arguments, $\* is equivalent to $1 $2. |
| 5 | **$@**   * All the arguments are individually double quoted. * If a script receives two arguments, $@ is equivalent to $1 $2. |
| 6 | **$?**   * The exit status of the last command executed. * As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful. * Some commands return additional exit statuses for particular reasons. * You got 127 that is the exit status of last executed command exited with some error (most probably). Commands on successful completion exit with an exit status of 0 (most probably). |
| 7 | **$$**   * The process number of the current shell. * For shell scripts, this is the process ID under which they are executing. |
| 8 | **$!** The process number of the last background command. |

* the "$\*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

**Array**

* array\_name=(value1 ... valuen)
* Accessing Array Values after you have set any array variable, you access it as follows − ${array\_name[index]}
* You can access all the items in an array in one of the following ways −
* ${array\_name[\*]}
* ${array\_name[@]}
* **#!/bin/sh**

**NAME[0]="Zara"**

**NAME[1]="Qadir"**

**NAME[2]="Mahnaz"**

**NAME[3]="Ayan"**

**NAME[4]="Daisy"**

**echo "First Index: ${NAME[0]}"**

**echo "Second Index: ${NAME[1]}"**

* **ArrayName=("element 1" "element 2" "element 3")**

**distro=("redhat" "debian" "gentoo")**

declare -a indices

indices=([0]="purple" [1]="reddish-orange" [2]="light green")

for((i=0; i < ${#indices[@]}; i++))

do

echo ${indices[${i}]}

done

* **bash$ Colors=([0]="purple" [1]="reddish-orange" [2]="light green")**

**bash$ echo ${Colors[@]}**

**purple reddish-orange light green**

* You can easily find out bash shell array length using following syntax:

**${#ArrayName[@]}**

If subscript is @ or \*, the word expands to all members of name. By prefixing # to variable you will find length of an array (i.e number of elements)

* **for (( i=0;i<$ELEMENTS;i++)); do**

**echo ${ARRAY[${i}]}**

**done**

**bc command**

* bc command is used for **command line calculator**. It is similar to basic calculator by using which we can do basic mathematical calculations.
* Arithmetic operations are the most basic in any kind of programming language. Linux or Unix operating system provides the bc command and expr command for doing arithmetic calculations. You can use these commands in bash or shell script also for evaluating arithmetic expressions.
* **d=$(echo "sqrt(( (2\*$i) - 1 - $j)^2 )" | bc)**
* **x=`echo "var=500;var%=7;var" | bc`**

**$ echo $x**

* bc is a command-line utility, not some obscure part of shell syntax. The utility reads mathematical expressions from its standard input and prints values to its standard output. **Since it is not part of the shell, it has no access to shell variables.**
* The shell pipe operator (|) connects the standard output of one shell command to the standard input of another shell command. For example, you could send an expression to bc by using the echo utility on the left-hand side of a pipe:

echo 2+2 | bc

This will print 4, since there is no more here than meets the eye.

So I suppose you wanted to do this:

a=2.77

b=2.0

for c in $(seq 0. 0.001 0.02); do

echo "$a \* $b \* $c" | bc

done

* Note: The expansion of the shell variables is happening when the shell processes the argument to echo, as you could verify by leaving off the bc:

a=2.77

b=2.0

for c in $(seq 0. 0.001 0.02); do

echo -n "$a \* $b \* $c" =

echo "$a \* $b \* $c" | bc

done

So bc just sees numbers.

* If you wanted to save the output of bc in a variable instead of sending it to standard output (i.e. the console), you could do so with normal command substitution syntax:

a=2.77

b=2.0

for c in $(seq 0. 0.001 0.02); do

d=$(echo "$a \* $b \* $c" | bc)

echo "$d"

done

* <https://www.geeksforgeeks.org/bc-command-linux-examples/>

**Operation and operator**

* Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either awk or expr.
* Bourne Shell supports the following relational operators that are specific to numeric values. **These operators do not work for string values unless their value is numeric.**
* For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".
* val=`expr 2 + 2`

echo "Total value : $val"

The above script will generate the following result −

Total value : 4

* **for assigning value we donnot use $ and for accesss the value we use $**
* The following points need to be considered while adding −
* There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
* The complete expression should be enclosed between ‘ ‘, called the backtick. Backtick is use to execute shell command.
* **#!/bin/bash**

**# use backticks " ` ` " to execute shell command**

**echo `uname -o`**

**# executing bash command without backticks**

**echo uname -o**

* The **uname** tool is most commonly used to determine the processor architecture, the system hostname and the version of the kernel running on the system.
* **`cmd` # Command Substitution**
* **$(cmd) # Command Substitution**

[quoting - When to use bash ANSI C style escape, e.g. $'\n' - Unix & Linux Stack Exchange](https://unix.stackexchange.com/questions/155367/when-to-use-bash-ansi-c-style-escape-e-g-n)

echo $(pwd) # --> /d/shell/shell/basic

echo `pwd` # --> /d/shell/shell/basic

echo pwd # --> pwd

echo 'pwd' # --> pwd

echo "pwd"  # --> pwd

# echo variable BASH\_VAR

echo $BASH\_VAR

echo $'$BASH\_VAR\nok'

echo 'a\nb' #--> a\nb

echo $'a\nb'

# --> a

# -->b

echo "a\nb" # --> a\nb

* Assume variable a holds 10 and variable b holds 20 then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / (Division) | Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| = (Assignment) | Assigns right operand in left operand | a = $b would assign value of b into a |
| == (Equality) | Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != (Not Equality) | Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |

* All the arithmetical calculations are done using long integers.

val=`expr $a + $b`

div=$(expr "$a" / "$b")

echo "Addition is $val"

echo "Multilpication is `expr $a \\* $b`"

echo “Multiplication is $(a \* b)”

echo “Multiplication is $(a\*b)”

echo "subtraction is $(expr "$a" - "$b")"

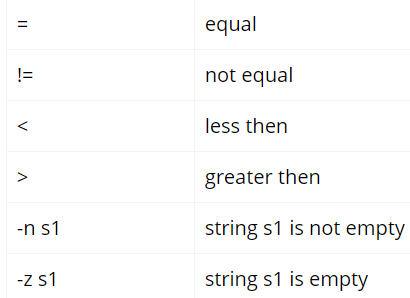
echo "Division is (`expr $a / $b`)"

* \* means "all files in the current directory". To instead mean a literal asterisk/multiplication character, you have to escape it:
* “”hello”” 🡪 hello “\”hello\”” 🡪 “hello”
* [We should use \ on the \* symbol for multiplication.](https://www.tutorialspoint.com/unix/unix-basic-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-eq** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a -eq $b ] is not true. |
| **-ne** | Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true. | [ $a -ne $b ] is true. |
| **-gt** | Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true. | [ $a -gt $b ] is not true. |
| **-lt** | Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true. | [ $a -lt $b ] is true. |
| **-ge** | Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true. | [ $a -ge $b ] is not true. |
| **-le** | Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true. | [ $a -le $b ] is true. |

|  |  |  |
| --- | --- | --- |
| **Operat or** | **Description** | **Example** |
| **!** | This is logical negation. This inverts a true condition into false and vice versa. | [ ! false ] is true. |
| **-o** | This is logical **OR**. If one of the operands is true, then the condition becomes true. | [ $a -lt 20 -o $b -gt 100 ] is true. |
| **-a** | This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false. | [ $a -lt 20 -a $b -gt 100 ] is false. |

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **=** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a = $b ] is not true. |
| **!=** | Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true. | [ $a != $b ] is true. |
| **-z** | Checks if the given string operand size is zero; if it is zero length, then it returns true. | [ -z $a ] is not true. |
| **-n** | Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true. | [ -n $a ] is not false. |
| **str** | Checks if **str** is not the empty string; if it is empty, then it returns false. | [ $a ] is not false. |



|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-b file** | Checks if file is a block special file; if yes, then the condition becomes true. | [ -b $file ] is false. |
| **-c file** | Checks if file is a character special file; if yes, then the condition becomes true. | [ -c $file ] is false. |
| **-d file** | Checks if file is a directory; if yes, then the condition becomes true. | [ -d $file ] is not true. |
| **-f file** | Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true. | [ -f $file ] is true. |
| **-g file** | Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true. | [ -g $file ] is false. |
| **-k file** | Checks if file has its sticky bit set; if yes, then the condition becomes true. | [ -k $file ] is false. |
| **-p file** | Checks if file is a named pipe; if yes, then the condition becomes true. | [ -p $file ] is false. |
| **-t file** | Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true. | [ -t $file ] is false. |
| **-u file** | Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true. | [ -u $file ] is false. |
| **-r file** | Checks if file is readable; if yes, then the condition becomes true. | [ -r $file ] is true. |
| **-w file** | Checks if file is writable; if yes, then the condition becomes true. | [ -w $file ] is true. |
| **-x file** | Checks if file is executable; if yes, then the condition becomes true. | [ -x $file ] is true. |
| **-s file** | Checks if file has size greater than 0; if yes, then condition becomes true. | [ -s $file ] is true. |
| **-e file** | Checks if file exists; is true even if file is a directory but exists. | [ -e $file ] is true. |

* All the conditional expressions should be inside square braces with spaces around them, for example [ $a == $b ] is correct whereas, [$a==$b] is incorrect. For example, [ $a <= $b ] is correct whereas, [$a <= $b] is incorrect.
* <https://www.tutorialspoint.com/unix/unix-basic-operators.htm>
* https://www.tutorialspoint.com/basic-operators-in-shell-scripting

**if else**

* if cp $source $destination//doubt
* if [ $num1 -gt $num2 ] is correct

if [$num1 -gt $num2] is not corrrect

[if [$num1-gt$num2] is not correct](https://www.tutorialspoint.com/unix/unix-basic-operators.htm)

[if [ $((dividend % divisior)) == 0 ]](https://www.tutorialspoint.com/unix/unix-basic-operators.htm) is correct

if [ $(($dividend % $divisior)) == 0 ] is correct

if [ $((dividend%divisior)) == 0 ] is correct

if [ $($dividend % $divisior) == 0 ] is not correct

**loop**

* select var in word1 word2 ... wordN

do

Statement(s) to be executed for every word.

done

* select var in {2..20..1}

do

Statement(s) to be executed for every word.

done

* **#!/bin/ksh**

**select DRINK in tea cofee water juice appe all none**

**do**

**case $DRINK in**

**tea|cofee|water|all)**

**echo "Go to canteen"**

**;;**

**juice|appe)**

**echo "Available at home"**

**;;**

**none)**

**break**

**;;**

**\*) echo "ERROR: Invalid selection"**

**;;**

**esac**

**done**

* **while command**

**do**

**Statement(s) to be executed if command is true**

**done**

* **until command**

**do**

**Statement(s) to be executed until command is true**

**done**

* while [ $i -le $num ] is correct

while [$i -le $num] is incorrect

while (( $i <= $num )) is correct

while (($i<=$num)) is correct

while (($i <= $num)) is correct

while ((i<=num)) is correct

while (($((i<=num)))) is correct

while ($((i<=num))) is incorrect

while $((i<=num)) is incorrect

* + - * **for val in {2..20..2}**

**do**

**echo "$val"**

**done**

* + - * **for val in 0 1 2 3 4 5 6 7 8 9 10**

**do**

**echo "$val"**

**done**

* + - * **for val in {0..20..2}**

**{**

**echo "$val"**

**}**

* + - * **for(( i = 1; i <= n; i++))**

**do**

**fact=$((fact \* i))**

**done**

* + - * **for(( i = 1; i <= n; i=`expr $i + 1`))**

**do**

**fact=$((fact \* i))**

**done**

* + - * **for((i=2;i<=n;i++))**

**{**

**fact=$((fact \* i)) #fact = fact \* i**

**}**

for f in $( ls );

do

    echo $f

done

* Cannot use while, until and select loop like for loop

**Echo**

* echo -e "Hello earth\n"
* echo "Hello sidd\n"#this line will not read \n as newline charactera as there is no -e
* echo -e "Hello user \c" #suppress trailing new line with backspace interpretor ‘-e‘ to continue without emitting new line.
* echo -e "How\v are\v you" #his option is used to create vertical tab spaces.
* **echo \***

#echo \* is similiar to ls

* echo -n "How as your day"#this will not echo trailing newline

**awk**

* **Awk is a scripting language** used for manipulating data and generating reports.
* The awk command programming language **requires no compiling**, and allows the user to use variables, numeric functions, string functions, and logical operators.
* Awk is abbreviated from the names of the developers – Aho, Weinberger, and Kernighan
* awk options 'selection \_criteria {action }' input-file > output-file
* **who**

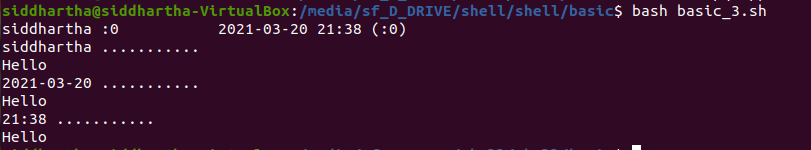
**for user in `who | awk '{print $1, $3, $4}'`**

**do**

**echo "$user ..........."**

**echo "Hello"**

**done**

****

* $0 represents the whole line.
* $ awk '{print}' employee.txt will print the whole text inside employee.txt

**Shell Programming Examples**

1- floyd's triangle

**echo "Input number of rows"**

**read num**

**n=1**

**echo -e "Floyd's triangle\n"**

**#-e is for evoluatte escape sequence**

**for(( i=1; i<=num; i++ ))**

**do**

**for(( j=1; j<=i; j++))**

**do**

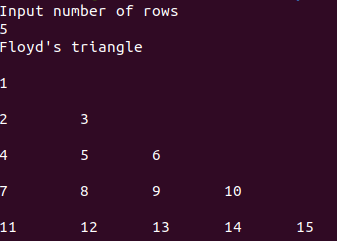
**echo -e -n "$n\t "**

**((n++))**

**done**

**echo -e " \n "**

**done**

****

2- Triangular Numbers are those numbers which are obtained by continued summation of the natural numbers 1, 2, 3, 4, 5, etc.

Triangular Number Example: 15 is Triangular Number because it can be obtained by 1+2+3+4+5+6 i.e. 1+2+3+4+5+6=15

**echo "Enter an integer"**

**read num**

**sum=0**

**flag=0**

**for((i=1; sum<=num ;i++))**

**do**

**((sum+=i))**

**if [ $sum -eq $num ]**

**then**

**flag=1**

**break**

**fi**

**done**

**if [ $flag -eq 1 ]**

**then**

**echo "$num is triangular number"**

**else**

**echo "$num is not triangular number"**

**fi**

3- A positive integer is called an Armstrong number (of order n) if

abcd... = a^n + b^n + c^n + d^n +

In the case of an Armstrong number of 3 digits, the sum of cubes of each digit is equal to the number itself.

For example, 153 is an Armstrong number because 153 = 1\*1\*1 + 5\*5\*5 + 3\*3\*3

**echo "Enter a nmber"**

**read num**

**temp\_num=$num**

**reminder=0**

**result=0**

**until (( $temp\_num == 0 ))**

**do**

**remainder = $(( $originalNum % 10 ))**

**result=$((($remainder \* $remainder \* $remainder) + $result))**

**temp\_num=$(( $temp\_num / 10 ))**

**done**

**if ($result == $num)**

**then**

**echo "$num is armstrong number"**

**else**

**echo "$num is not armstrong number"**

**fi**

4- In mathematics, an automorphic number is a number whose square "ends" in the same digits as the number itself. For example, 52 = 25, 62 = 36, 762 = 5776, and 8906252 = 793212890625, so 5, 6, 76 and 890625 are all automorphic numbers.

**check\_automorphic\_number(){**

**temp=$1**

**length=${#temp}**

**divisor=$(echo "10^$length" | bc)**

**temp=$(echo "$1^2" | bc)**

**temp=$(( $temp%$divisor ))**

**if(( $temp == $1 ))**

**then**

**echo "$1 is automorphic number"**

**return**

**fi**

**echo "$1 is not automorphic number"**

**}**

5-An abundant Number (also known as excessive number) is a number in the number theory which itself is smaller than the sum of all its proper divisors. For example,12 is an abundant Number : divisors 1,2,3,4,6 , sum =16 >12.

The difference between the sum of divisors and the number is called abundance

**check\_abundant\_number(){**

**temp=0**

**for(( i=1; $i<=($1/2); i++))**

**do**

**if(( ($1 % $i)==0))**

**then**

**temp=$(( $temp+$i))**

**fi**

**done**

**if(($temp>=$1))**

**then**

**echo "$1 is an abundant number"**

**return**

**fi**

**echo "$1 is not an abundant number"**

**}**

1. I/O
2. if-then-fi, if-then–else-fi, if-then-elif-else-fi
3. Case
4. Loops

#Bash Scripting Examples…….

#.............................................................................

#

echo "Input two integer numbers"

read a

read b

c=` expr $a + $b `

echo "The sum is ", $c

echo "The sum of $a and $b is $c \n"

#.............................................................................

echo "Input two real numbers"

read a

read b

c=$(echo "scale=2; $a / $b" | bc)

echo "The sum of $a and $b is $c \n"

#.............................................................................

if mv testing.sh test.sh

then

echo File is renamed successfully

fi

#.............................................................................

if mv testing.sh test.sh

then

echo File is renamed successfully

else

echo File is not renamed successfully

fi

#.............................................................................

echo input two integers

read a b

if [ $a -gt $b ]

then

echo "$a is greater"

else

echo "$b is greater"

fi

#.............................................................................

echo input two integers

read a b

if ( $a > $b )

then

echo "$a is greater"

else

echo "$b is greater"

fi

#.............................................................................

echo "Input an integer"

read num

if [ $num -gt 0 ]

then

echo "Number is positive"

elif [ $num -eq 0 ]

then

echo "Number is zero"

else

echo "Number is negative"

fi

#.............................................................................

#odd even example

echo "Input an integer"

read num

rem=`expr $num % 2`

if [ $rem -eq 0 ]

then

echo "Number is even"

else

echo "Number is odd"

fi

#.............................................................................

# Nested if

if [ condition1 ]

then

if [ condition11 ]

then

....

else

....

fi

else

if [ condition12 ]

then

....

else

....

fi

fi

# Nested if example

# Nested if example

echo input three integers

read a b c

if [ $a -gt $b ]

then

if [ $a -gt $c ]

then

echo a=$a is the greatest

else

echo c=$c is the greatest

fi

else

if [ $b -gt $c ]

then

echo b=$b is the greatest

else

echo c=$c is the greatest

fi

fi

#.............................................................................

echo input three integers

read a b c

if [ $a -gt $b -a $a -gt $c ]

then

echo a=$a is the greatest

fi

if [ $b -gt $a -a $b -gt $c ]

then

echo b=$b is the greatest

fi

if [ $c -gt $a -a $c -gt $b ]

then

echo c=$c is the greatest

fi

#.............................................................................

echo input three integers

read a b c

if [ $a -gt $b ] && [ $a -gt $c]

then

echo a=$a is the greatest

fi

if [ $b -gt $a ] && [ $b -gt $c ]

then

echo b=$b is the greatest

else

echo c=$c is the greatest

fi

#.............................................................................

echo input two integers

read a b

echo Input the choice of arithmetic operation

read choice

case $choice in

"+") c=`expr $a + $b`

;;

'-') c=`expr $a - $b`

;;

"\*") c=`expr $a \\* $b`

;;

"/") c=`expr $a / $b`

;;

\*) c=`expr $a + $b`

;;

esac

#

echo Input N

read n

sum=0

for ((i=1;$i<=$n;i+=1))

do

sum=`expr $sum + $i`

done

echo The sum of N=$n Natural Numbers is $sum

#.............................................................................

echo "Input an integer"

read num

sum=0

for (( i=1 ; i<=num ; i+=2 ))

do

sum=`expr $sum + $i`

done

echo The odd series sum is $sum

#.............................................................................

echo "Input the number n"

read n

for i in 1 2 3 4 5

do

echo $i

done

for ((i=1 ; i<=n ; i++))

do

echo $i

done

#

#.............................................................................

echo Input N

read n

sum=0

for ((i=2;$i<=$n;i=`expr $i + 2`))

do

sum=`expr $sum + $i`

done

echo The sum of even numbers from 2 to $n is $sum

#.............................................................................

echo "Input an integer to print the pattern of numbers"

read n

echo -e "\n The Pattern is ......\n"

for((i=1; $i<=$n ;i++))

do

for((j=1;$j<=$i;j++))

do

echo -n $j " "

done

echo -e "\n"

done

#.............................................................................

echo "Input the number n"

read n

i=1

while [ $i -le $n ]

do

echo $i

i=`expr $i + 1`

done

#.............................................................................

echo "Input an integer"

read num

sum=0

i=1

while [ $i -le $num ]

do

sum=`expr $sum + $i`

i=`expr $i + 1`

done

echo The odd series sum is $sum

#.............................................................................

#

echo "Input the number n"

read n

i=0

reverse=0

num=$n

while [ $num -gt 0 ]

do

reverse=`expr $reverse \\* 10 + $num % 10`

num=`expr $num / 10`

done

echo Reverse of $n is $reverse

#.............................................................................

#

echo "Input an integer"

read num

sum=0

i=1

until [ $i -gt $num ]

do

sum=`expr $sum + $i`

i=`expr $i + 1`

done

echo The odd series sum is $sum

#.............................................................................

#

echo Input the Number N

read n

i=0

while [ $i -le $n ]

do

j=$i

while [ $j -ge 0 ]

do

echo -n $j " "

j=`expr $j - 1`

done

echo " "

i=`expr $i + 1`

done

#.............................................................................

#

#.............................................................................

#

echo "Input the choice of operation"

echo -e "1:Factorial \n2:Palindrome ?\n3:Fibonacci\n4:Prime ?\n"

read choice

case $choice in

1) echo Input an integer to compute its factorial

read n

fact=1

for((i=1;$i<=$n;i++))

do

fact=`expr $fact \\* $i`

done

echo "Factorial is" , $fact

;;

2)

echo "Input the number n"

read n

i=0

reverse=0

num=$n

while [ $num -gt 0 ]

do

reverse=`expr $reverse \\* 10 + $num % 10`

num=`expr $num / 10`

done

if [ $reverse -eq $n ]

then

echo "The given integer $n is a Palindrome"

else

echo "The given integer $n is NOT a Palindrome"

fi

;;

3) echo "Input an integer"

read n

echo "Fibonacci Series is "

a=1

b=1

echo -n $a $b

for((i=3;$i<=n;i++))

do

c=`expr $a + $b`

echo -n $c " "

a=$b

b=$c

done

;;

4) echo "Prime Number ?"

echo Input an integer

read n

sqrtN=$(echo "sqrt($n)" | bc )

echo $sqrtN

;;

\*) echo "Invalid Choice"

;;

esac

#................................................................................................................................

array=`ls`

len=${#array[@]}

echo ${#array[\*]}

echo .... $len.....

for i in $array

do

echo $i

done

#.............................................................................

#

#.............................................................................

#

#case example

echo "Input an integer"

read choice

case $choice in

1) echo "Hai..."

;;

2) echo "hello..." ;;

\*) echo "Welcome.."

esac

#.............................................................................

echo "Input an integer"

read num

sum=0

for (( i=1 ; i<=num ; i+=2 ))

do

sum=`expr $sum + $i`

done

echo The odd series sum is $sum

#.............................................................................

#for example – to print file types in the current working directory

for file in `ls`

do

if [ -d $file ]

then

echo $file is a directory

elif [ -c $file ]

then

echo $file is a character device file

elif [ -b $file ]

then

echo $file is a block device file

elif [ -S $file ]

then

echo $file is a domain socket

elif [ -p $file ]

then

echo $file is a named pipe

elif [ -L $file ]

then

echo $file is a symbolic link

elif [ -f $file ]

then

echo $file is a regular file

else

echo $file

fi

done

#.........................................Triangular Number...................................…..

echo "Input an Integer"

read num

sum=0

flag=0

for((i=1;sum<=num;i++))

do

sum=$((sum+i))

if [ $sum -eq $num ]

then

flag=1

break

fi

done

if [ $flag -eq 1 ]

then

echo "$num is a Triangular number"

else

echo "$num is not a Triangular number"

fi

#......................................................................................................................

while((1))

do

fact=1

echo "Input an integer(0 to stop iterating)"

read num

if ((num<0))

then

continue

fi

if ((num==0))

then

break

fi

for((i=1;i<=num;i++))

do

((fact\*=i))

done

echo "Factorial of $num is $fact"

done

#....................................Floyd’s Triangle…………………..

echo "Input the number of rows in Floyd's Triangle"

read n

num=1

for (( i = 1 ;i<=n; i++))

do

for (( j = 1 ; j <= i ;j++))

do

echo -n $num " "

((num++))

done

echo -e "\n"

done

#..........................Command Line Arguments …………………

echo "The number of commandline arguments passed is $#"

echo "The name of the shell script is $0"

for i in $@

do

echo $i

done

#.............. A sample run the above script as given below ……………….…

bash main.sh 11 22 33 44 55

The number of commandline arguments passed is 5

The name of the shell script is main.sh

11

22

33

44

55

#.................................Write message to who users...............................................................

for user in `who | awk ‘{print $1}’`

do

echo $user

write $user

done

#...........................continue 2 example ..............................

for((;;))

do

i=1

while ((1 ))

do

((i++))

if (( i % 5 == 0 ))

then

continue 2

fi

echo $i

read

done

done